Writing a system call tracer using eBPF

SH4DY 2024-08-03 | eBPF, linux, low-level

Pre-Requisites

System calls, eBPF, C, basics of low-level programming.

Introduction

eBPF (Extended Berkeley Packet Filter) is a technology that allows users to run custom programs within the kernel. BPF / or cBPF (classic BPF), the predecessor of eBPF provided a simple and efficient way to filter packets based on predefined rules. eBPF programs offer enhanced safety, portability, and maintainability as compared to kernel modules. There are several high-level methods available for working with eBPF programs, such as Cilium's go library, bpftrace, libbpf, etc.

- `Note`: This post requires the reader to have a basic understanding of `eBPF`. If you're not familiar with it, this post by `ebpf.io` is a great read.

Objectives

You must already be familiar with the famous tool `strace`. We'll be developing something similar to that using eBPF. For example,

```
./beetrace /bin/ls
```

Concepts

Before we start writing our tool, we need to familiarize ourselves with some key concepts.

1. `Tracepoints`: They are instrumentation points placed in various parts of the Linux kernel code. They provide a way to hook into specific events or code paths within the kernel without modifying the kernel source code. The events available of tracing can be found at `/sys/kernel/debug/tracing/events`.
2. The `SEC` macro: It creates a new section with the name as the name of the tracepoint within the target ELF. For example, `SEC(tracepoint/raw_syscalls/sys_enter)` creates a new section with this name. The sections can be viewed using readelf.

```
readelf -s --wide somefile.o
```

3. `Maps`: They are shared data structures that can be accessed from both eBPF programs and applications running in the userspace.

Writing the eBPF programs

We won’t be writing a comprehensive tool for tracing all the system calls due to the vast number of system calls present in the Linux kernel. Instead, we’ll focus on tracing a few common system calls. To achieve this, we’ll write two types of programs: eBPF programs and a loader (which loads the BPF objects into the kernel and attaches them).

Let’s start by creating a few data structures to set things up.

```
// controller.h

// SYS_ENTER : for retrieving system call arguments
// SYS_EXIT : for retrieving the return values of syscalls

typedef enum
{
    SYS_ENTER,
    SYS_EXIT
} event_mode;

struct inner_syscall_info
{
    union
```

```
    {
        struct
        {
            // For SYS_ENTER mode
            char name[32];
            int num_args;
            long syscall_nr;
            void *args[MAX_ARGS];
        };
        long retval; // For SYS_EXIT mode
    };
    event_mode mode;
};

struct default_syscall_info{
    char name[32];
    int num_args;
};

// Array for storing the name and argument count of system calls
const struct default_syscall_info syscalls[MAX_SYSCALL_NR] = {
    [SYS_fork] = {"fork", 0},
    [SYS_alarm] = {"alarm", 1},
    [SYS_brk] = {"brk", 1},
    [SYS_close] = {"close", 1},
    [SYS_exit] = {"exit", 1},
    [SYS_exit_group] = {"exit_group", 1},
    [SYS_set_tid_address] = {"set_tid_address", 1},
    [SYS_set_robust_list] = {"set_robust_list", 1},
    [SYS_access] = {"access", 2},
    [SYS_arch_prctl] = {"arch_prctl", 2},
    [SYS_kill] = {"kill", 2},
    [SYS_listen] = {"listen", 2},
    [SYS_munmap] = {"sys_munmap", 2},
    [SYS_open] = {"open", 2},
    [SYS_stat] = {"stat", 2},
    [SYS_fstat] = {"fstat", 2},
    [SYS_lstat] = {"lstat", 2},
    [SYS_accept] = {"accept", 3},
    [SYS_connect] = {"connect", 3},
    [SYS_execve] = {"execve", 3},
    [SYS_ioctl] = {"ioctl", 3},
    [SYS_getrandom] = {"getrandom", 3},
    [SYS_lseek] = {"lseek", 3},
    [SYS_poll] = {"poll", 3},
    [SYS_read] = {"read", 3},
    [SYS_write] = {"write", 3},
```

```
    [SYS_mprotect] = {"mprotect", 3},
    [SYS_openat] = {"openat", 3},
    [SYS_socket] = {"socket", 3},
    [SYS_newfstatat] = {"newfstatat", 4},
    [SYS_pread64] = {"pread64", 4},
    [SYS_prlimit64] = {"prlimit64", 4},
    [SYS_rseq] = {"rseq", 4},
    [SYS_sendfile] = {"sendfile", 4},
    [SYS_socketpair] = {"socketpair", 4},
    [SYS_mmap] = {"mmap", 6},
    [SYS_recvfrom] = {"recvfrom", 6},
    [SYS_sendto] = {"sendto", 6},
};
```

The loader will read the path of the ELF file to be traced, which will be provided by the user as a command line argument. Then, the loader will spawn a child process and use `execve` to run the program specified in the command line argument.

The parent process will handle all the necessary setup for loading and attaching the eBPF programs. It also performs the crucial task of sending the child process's ID to the eBPF program via the BPF hashmap.

```
// loader.c

int main(int argc, char **argv)
{
  if (argc < 2)
  {
    fatal_error("Usage: ./beetrace <path_to_program>");
  }

  const char *file_path = argv[1];

  pid_t pid = fork();
  if (pid == 0)
  {
    // Child process
    int fd = open("/dev/null", O_WRONLY);
    if(fd==-1){
        // error
    }
    dup2(fd, 1); // disable stdout for the child process
    sleep(2); // wait for the parent process to do the required setup for t
    execve(file_path, NULL, NULL);
```

```
  }
  else{
    // Parent process
  }
}
```

To trace system calls, we need to write eBPF programs that are triggered by the `tracepoint/raw_syscalls/sys_enter` and `tracepoint/raw_syscalls/sys_exit` tracepoints. These tracepoints provide access to the system call number and arguments. For a given system call, the `tracepoint/raw_syscalls/sys_enter` tracepoint is always triggered before the `tracepoint/raw_syscalls/sys_exit` tracepoint. We can use the former to retrieve the system call arguments and the latter to obtain the return value. Additionally, we will use eBPF maps to share information between the user-space program and our eBPF programs. Specifically, we will use two types of eBPF maps: hashmaps and ring buffers.

```
// controller.c

// Hashmap
struct
{
  __uint(type, BPF_MAP_TYPE_HASH);
  __uint(key_size, 10);
  __uint(value_size, 4);
  __uint(max_entries, 256 * 1024);
} pid_hashmap SEC(".maps");

// Ring buffer
struct
{
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, 256 * 1024);
} syscall_info_buffer SEC(".maps");

```

Having defined the maps, we're ready to write the programs. Let's start by writing the program for the tracepoint `tracepoint/raw_syscalls/sys_enter`.

```
// loader.c

SEC("tracepoint/raw_syscalls/sys_enter")
int detect_syscall_enter(struct trace_event_raw_sys_enter *ctx)
```

```
{
  // Retrieve the system call number
  long syscall_nr = ctx->id;
  const char *key = "child_pid";
  int target_pid;

  // Reading the process id of the child process in userland
  void *value = bpf_map_lookup_elem(&pid_hashmap, key);
  void *args[MAX_ARGS];

  if (value)
  {
    target_pid = *(int *)value;

    // PID of the process that executed the current system call
    pid_t pid = bpf_get_current_pid_tgid() & 0xffffffff;
    if (pid == target_pid && syscall_nr >= 0 && syscall_nr < MAX_SYSCALL_NR
    {

      int idx = syscall_nr;
      // Reserve space in the ring buffer
      struct inner_syscall_info *info = bpf_ringbuf_reserve(&syscall_info_b
      if (!info)
      {
        bpf_printk("bpf_ringbuf_reserve failed");
        return 1;
      }

      // Copy the syscall name into info->name
      bpf_probe_read_kernel_str(info->name, sizeof(syscalls[syscall_nr].nam
      for (int i = 0; i < MAX_ARGS; i++)
      {
        info->args[i] = (void *)BPF_CORE_READ(ctx, args[i]);
      }
      info->num_args = syscalls[syscall_nr].num_args;
      info->syscall_nr = syscall_nr;
      info->mode = SYS_ENTER;
      // Insert into ring buffer
      bpf_ringbuf_submit(info, 0);
    }
  }
  return 0;
}
```

Similarly, we can write the program for reading the return value and sending it to userland.

```
// controller.c

SEC("tracepoint/raw_syscalls/sys_exit")
int detect_syscall_exit(struct trace_event_raw_sys_exit *ctx)
{
  const char *key = "child_pid";
  void *value = bpf_map_lookup_elem(&pid_hashmap, key);
  pid_t pid, target_pid;

  if (value)
  {
    pid = bpf_get_current_pid_tgid() & 0xffffffff;
    target_pid = *(pid_t *)value;
    if (pid == target_pid)
    {
      struct inner_syscall_info *info = bpf_ringbuf_reserve(&syscall_info_b
      if (!info)
      {
        bpf_printk("bpf_ringbuf_reserve failed");
        return 1;
      }
      info->mode = SYS_EXIT;
      info->retval = ctx->ret;
      bpf_ringbuf_submit(info, 0);
    }
  }
  return 0;
}
```

Let’s now finalize the functionality for the parent process in the loader program. Before doing that, we need to understand how some key functions work.

1. `bpf_object__open`: Creates a bpf_object by opening the BPF ELF object file pointed to by the passed path and loading it into memory.

```
LIBBPF_API struct bpf_object *bpf_object__open(const char *path);
```

2. `bpf_object__load`: Loads BPF object into kernel.

```
LIBBPF_API int bpf_object__load(struct bpf_object *obj);
```

3. `bpf_object__find_program_by_name`: Returns a pointer to a valid BPF program.

```
LIBBPF_API struct bpf_program *bpf_object__find_program_by_name(const struct
```

4. `bpf_program__attach`: Function for attaching a BPF program based on auto-detection of program type, attach type, and extra paremeters, where applicable.

```
LIBBPF_API struct bpf_link *bpf_program__attach(const struct bpf_progra
```

5. `bpf_map__update_elem`: Allows to insert or update value in BPF map that corresponds to provided key.

```
LIBBPF_API int bpf_map__update_elem(const struct bpf_map *map,const voi
```

6. `bpf_object__find_map_fd_by_name`: Given a BPF map name, it returns a file descriptor to it.

```
LIBBPF_API int bpf_object__find_map_fd_by_name(const struct bpf_object
```

7. `ring_buffer__new`: Returns a pointer to the ring buffer.

```
LIBBPF_API struct ring_buffer *ring_buffer__new(int map_fd, ring_buffer
```

The second argument must be a function which can be used for handling the data received from the ring buffer.

```
bool initialized = false;

static int syscall_logger(void *ctx, void *data, size_t len)
{
```

```
  struct inner_syscall_info *info = (struct inner_syscall_info *)data;
  if (!info)
  {
    return -1;
  }

  if (info->mode == SYS_ENTER)
  {
    initialized = true;
    printf("%s(", info->name);
    for (int i = 0; i < info->num_args; i++)
    {
      printf("%p,", info->args[i]);
    }
    printf("\b) = ");
  }
  else if (info->mode == SYS_EXIT)
  {
    if (initialized)
    {
      printf("0x%lx\n", info->retval);
    }
  }
  return 0;
}
```

It prints the name and arguments of the system calls.

8. `ring_buffer__consume`: It processes the available events in the ring buffer.

```
LIBBPF_API int ring_buffer__consume(struct ring_buffer *rb);
```

We now have everything needed to write the loader.

```
// loader.c
#include <bpf/libbpf.h>
#include "controller.h"
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
```

```c
#include <unistd.h>

void fatal_error(const char *message)
{
  puts(message);
  exit(1);
}

bool initialized = false;

static int syscall_logger(void *ctx, void *data, size_t len)
{
  struct inner_syscall_info *info = (struct inner_syscall_info *)data;
  if (!info)
  {
    return -1;
  }

  if (info->mode == SYS_ENTER)
  {
    initialized = true;
    printf("%s(", info->name);
    for (int i = 0; i < info->num_args; i++)
    {
      printf("%p,", info->args[i]);
    }
    printf("\b) = ");
  }
  else if (info->mode == SYS_EXIT)
  {
    if (initialized)
    {
      printf("0x%lx\n", info->retval);
    }
  }
  return 0;
}

int main(int argc, char **argv)
{
  int status;
  struct bpf_object *obj;
  struct bpf_program *enter_prog, *exit_prog;
  struct bpf_map *syscall_map;
  const char *obj_name = "controller.o";
  const char *map_name = "pid_hashmap";
  const char *enter_prog_name = "detect_syscall_enter";
```

```
    const char *exit_prog_name = "detect_syscall_exit";
    const char *syscall_info_bufname = "syscall_info_buffer";

    if (argc < 2)
    {
      fatal_error("Usage: ./beetrace <path_to_program>");
    }
    const char *file_path = argv[1];

    pid_t pid = fork();
    if (pid == 0)
    {
      int fd = open("/dev/null", O_WRONLY);
      if(fd==-1){
        fatal_error("failed to open /dev/null");
      }
      dup2(fd, 1);
      sleep(2);
      execve(file_path, NULL, NULL);
    }
    else
    {
      printf("Spawned child process with a PID of %d\n", pid);
      obj = bpf_object__open(obj_name);
      if (!obj)
      {
        fatal_error("failed to open the BPF object");
      }
      if (bpf_object__load(obj))
      {
        fatal_error("failed to load the BPF object into kernel");
      }

      enter_prog = bpf_object__find_program_by_name(obj, enter_prog_name);
      exit_prog = bpf_object__find_program_by_name(obj, exit_prog_name);

      if (!enter_prog || !exit_prog)
      {
        fatal_error("failed to find the BPF program");
      }
      if (!bpf_program__attach(enter_prog) || !bpf_program__attach(exit_prog
      {
        fatal_error("failed to attach the BPF program");
      }
      syscall_map = bpf_object__find_map_by_name(obj, map_name);
      if (!syscall_map)
      {
```

```
      fatal_error("failed to find the BPF map");
    }
    const char *key = "child_pid";
    int err = bpf_map__update_elem(syscall_map, key, 10, (void *)&pid, siz
    if (err)
    {
      printf("%d", err);
      fatal_error("failed to insert child pid into the ring buffer");
    }

    int rbFd = bpf_object__find_map_fd_by_name(obj, syscall_info_bufname);

    struct ring_buffer *rbuffer = ring_buffer__new(rbFd, syscall_logger, N

    if (!rbuffer)
    {
      fatal_error("failed to allocate ring buffer");
    }

    if (wait(&status) == -1)
    {
      fatal_error("failed to wait for the child process");
    }

    while (1)
    {
      int e = ring_buffer__consume(rbuffer);
      if (!e)
      {
        break;
      }
      sleep(1);
    }
  }
  return 0;
}
```

And, here are the eBPF programs. The C code will be compiled into a single object file.

```
// controller.c

#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
```

```
#include <bpf/bpf_core_read.h>
#include <sys/syscall.h>
#include "controller.h"

struct
{
  __uint(type, BPF_MAP_TYPE_HASH);
  __uint(key_size, 10);
  __uint(value_size, 4);
  __uint(max_entries, 256 * 1024);
} pid_hashmap SEC(".maps");

struct
{
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, 256 * 1024);
} syscall_info_buffer SEC(".maps");


SEC("tracepoint/raw_syscalls/sys_enter")
int detect_syscall_enter(struct trace_event_raw_sys_enter *ctx)
{
  // Retrieve the system call number
  long syscall_nr = ctx->id;
  const char *key = "child_pid";
  int target_pid;

  // Reading the process id of the child process in userland
  void *value = bpf_map_lookup_elem(&pid_hashmap, key);
  void *args[MAX_ARGS];

  if (value)
  {
    target_pid = *(int *)value;

    // PID of the process that executed the current system call
    pid_t pid = bpf_get_current_pid_tgid() & 0xffffffff;
    if (pid == target_pid && syscall_nr >= 0 && syscall_nr < MAX_SYSCALL_NR
    {

      int idx = syscall_nr;
      // Reserve space in the ring buffer
      struct inner_syscall_info *info = bpf_ringbuf_reserve(&syscall_info_b
      if (!info)
      {
        bpf_printk("bpf_ringbuf_reserve failed");
        return 1;
```

```
      }

      // Copy the syscall name into info->name
      bpf_probe_read_kernel_str(info->name, sizeof(syscalls[syscall_nr].nam
      for (int i = 0; i < MAX_ARGS; i++)
      {
        info->args[i] = (void *)BPF_CORE_READ(ctx, args[i]);
      }
      info->num_args = syscalls[syscall_nr].num_args;
      info->syscall_nr = syscall_nr;
      info->mode = SYS_ENTER;
      // Insert into ring buffer
      bpf_ringbuf_submit(info, 0);
    }
  }
  return 0;
}

SEC("tracepoint/raw_syscalls/sys_exit")
int detect_syscall_exit(struct trace_event_raw_sys_exit *ctx)
{
  const char *key = "child_pid";
  void *value = bpf_map_lookup_elem(&pid_hashmap, key);
  pid_t pid, target_pid;

  if (value)
  {
    pid = bpf_get_current_pid_tgid() & 0xffffffff;
    target_pid = *(pid_t *)value;
    if (pid == target_pid)
    {
      struct inner_syscall_info *info = bpf_ringbuf_reserve(&syscall_info_b
      if (!info)
      {
        bpf_printk("bpf_ringbuf_reserve failed");
        return 1;
      }
      info->mode = SYS_EXIT;
      info->retval = ctx->ret;
      bpf_ringbuf_submit(info, 0);
    }
  }
  return 0;
}

char LICENSE[] SEC("license") = "GPL";

```

Before compiling, we can create a test program which will be traced by our tool.

```
#include<stdio.h>
int main(){
    puts("tracer in action");
    return 0;
}
```

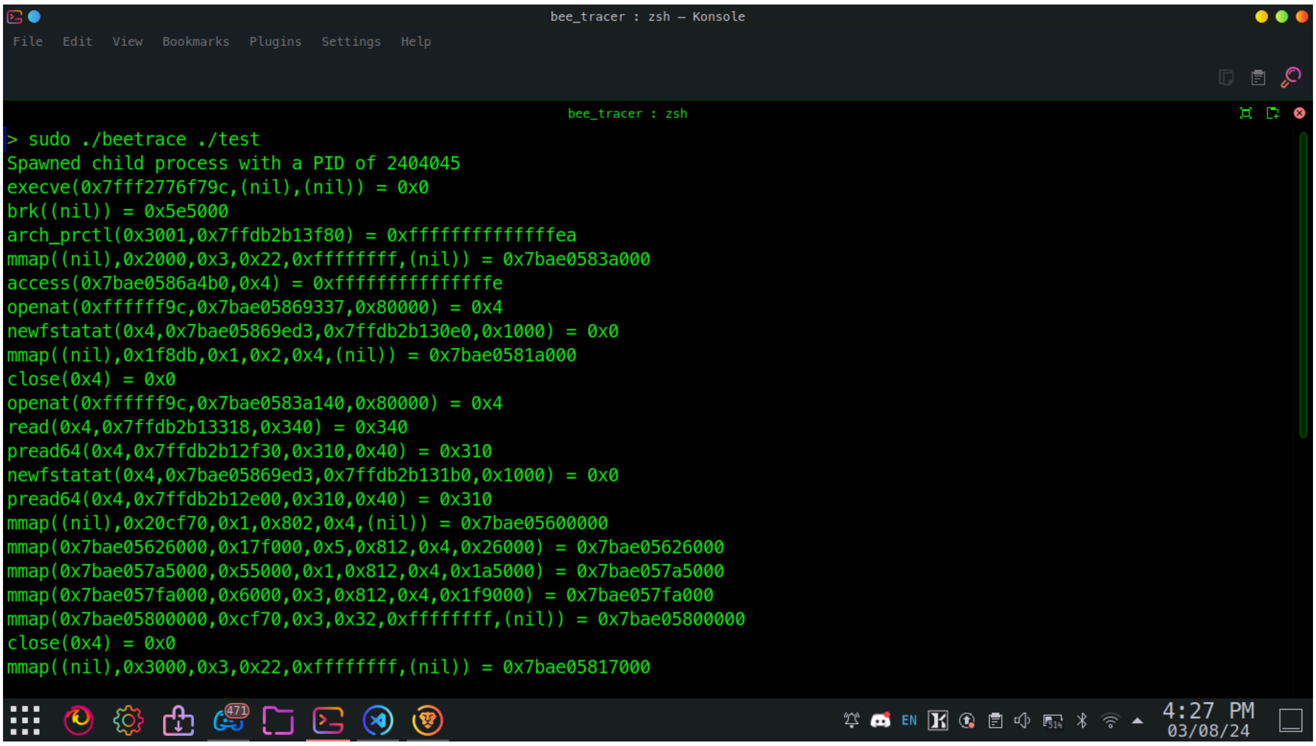
The following Makefile can be used to compile all the stuff.

```
compile:
        clang -O2 -g -Wall -I/usr/include -I/usr/include/bpf -o beetrace loa
        clang -O2 -g -target bpf -c controller.c -o controller.o

```

Now let’s execute the loader with root privileges.

```
sudo ./beetrace ./test
```



 Home | About | Blogs | Projects

The entire code can be found in this GitHub repository.

References:

https://ebpf.io/

https://github.com/libbpf/libbpf